Code Verification for Phenomenological Modeling and Simulation Software*

Patrick Knupp
Dept. 1414, Applied Mathematics and Applications
Sandia National Laboratories

1. Introduction

Code verification is an important activity within V&V. The latter consists of two basic parts, Verification and Validation. The end-product in Validation is an assessment of fidelity of the modeling and simulation software to the experimental data. Validation is supported by Verification, whose end-product is an assessment of both the software and the numerical solutions it produces. Verification is usually divided into two parts: Code Verification (CV) and Calculation (or Solution) Verification (SV). The end-product in CV is an assessment of the software, code, or codes that will be used in the V&V exercise, while the end-product of SV is an assessment of the particular numerical solutions that are to be compared to the experiment. At a minimum, the code assessment is made as it relates to the V&V exercise, but broader goals are often included.

The ASME gives a high-level definition of code verification:

> *Activities that "establish confidence, through the collection of evidence, that the mathematical model and solution algorithms are working correctly".*

This definition is supplemented by a guide [1] that provides a lengthy description of code verification. Although the guide is an excellent resource, those engaged in verification do not express a unanimous opinion on the end-products of code verification. The use of the ambiguous phrase 'working correctly' is appropriate given the lack of agreement. In addition, there are important research issues in code verification that remain to be settled. The intention of this chapter is to provide context for Code Verification and to describe the main concepts and practices. The chapter does not try to resolve the debate over the meaning of 'working correctly'.

V&V enhances the use of a computer code in order to simulate particular experiments of interest. Usually, the code that has been selected for a given V&V exercise is already well-developed, and many of the concepts and practices of Software Engineering (SE) have been applied [2]. Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software. Software Engineering encompasses knowledge, tools, and methods for defining software requirements, software design, computer programming, user interface design, software testing, and maintenance. Development of *requirements* is a critical early stage in software development and takes into account such things as the platforms on which the

*Sandia is a multiprogram laboratory operation by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

software will run, the input/output specifications, speed and memory requirements, code maintenance, version control and tracking, and many other concerns. A major activity in providing demonstrations that the requirements have been met is software testing. Software can be tested to demonstrate, for example, that the code will compile on all the required platforms, that all components are properly linked and interact correctly, that speed and memory requirements are met, that the expected behavior occurs, that errors and exceptions are properly handled, that the answer today is the same as the answer yesterday (unless it was intentionally changed), that the expected answer is produced, and many other testing goals. Testing methods include unit and functional testing, integration, stress, performance, memory, portability, and regression tests [3], [4].

Software Engineering applies to all software projects whether the application is business, banking, medical, engineering, or gaming. Different aspects of software engineering can be emphasized, depending on the requirements of a given software development project. Quoting from the abstract of [2],

Software [engineering] processes determine whether the development products of a given activity conform to the requirements of that activity and whether the software satisfies its intended use...[1]

Relating this statement to software that is selected for development and use within a V&V exercise, the software should conform not only to the requirements of the software development project from which it is obtained, but also to its intended use within V&V, namely, to compute asymptotically converged numerical solutions (along with error estimates and error bars) to the governing PDEs so that they can be compared to the physical experiments. It is this additional set of SE requirements that code verification, as understood by the V&V community, is primarily concerned.

To help delineate the boundaries of code verification, it is useful to mention the activity which immediately follows, namely calculation verification. Recall that in V&V, experimental data is compared against numerical solutions resulting from a set of computer simulations. Calculation verification provides an assessment of the numerical solutions resulting from these simulations, in order to make a proper comparison to the experimental data. The main goal of calculation verification is to estimate the discretization error and discretization error bars. It is important to note that in calculation verification the exact solution is always unknown, whereas in code verification many tests employ known solutions.

2. Code Verification Concepts

This discussion is focused on codes which solve partial differential equations for heat transfer, fluid flow, and chemical kinetics found in chemical propulsion and energetics modeling, although much of it also applies to other PDE-solving codes. Codes solving simpler mathematical models, such as ODEs, discrete models, networks, sets of equations, may require different types of tests than PDE-codes, but many of the topics covered here may apply to these cases as well.

An important fact about code verification is that it is not necessary that the mathematical model be an adequate representation of physical reality. The function of CV within V&V is simply to ensure that the numerical algorithms that solve the mathematical model (physically correct or not) are working properly, so that one is not mislead when using the code to assist in validating the

[1] It is an unfortunate fact that the terms verification and validation are used within the V&V community in a manner which is not entirely consistent with the way these terms are used in the IEEE standard. The present essay chooses to use the terms as they are understood by the V&V community since that is the primary audience.

model predictions against experimental data.

The mathematical model and its software implementation is an input to the code verification process. In order to perform CV, one must know the details of the mathematical model so that its implementation can be appropriately tested. It is usually not enough to say that the mathematical model is, for example, the heat equation. Rather, one must know detailed descriptions of the governing equations, boundary and initial conditions, assumptions on the modeled domain, PDE coefficients (including material models), and source terms. Additionally, the mathematical model includes the data or input associated with the values of the parameters and coefficients of the model. Ideally, this type of information is readily available.

Numerical analysis and numerical algorithms are employed to solve partial differential equations on a computer. Numerical analysis is the mathematical study of methods for discretizing PDEs to achieve consistency, stability, convergence, and other important properties such as discretization error and order-of-accuracy. Numerical algorithms are sets of instructions describing the specific computations one must perform to solve the PDEs. It is often the case, in developing PDE-solving software, that it is unclear in advance as to which of several possible discretizations and numerical algorithms is best (in terms of accuracy, efficiency, and robustness) to solve the given mathematical model, particularly with regard to a challenging application. Therefore, we make an important distinction at this point between two activities that relate to code verification:

- *Numerical Algorithm Property Verification (NAV)*: Determining whether the relevant numerical algorithms within a code have been implemented in agreement with their proven mathematical properties such as order-of-accuracy, stability, and consistency.
- *Numerical Algorithm Adequacy Verification (NAA)*: Determining whether relevant numerical algorithms within a code satisfy the accuracy, robustness, and speed requirements of their intended use.

Numerical algorithm property verification is concerned with whether the implementation in the code is consonant with the theoretically-proved algorithm properties (if any). This can be determined by comparing the numerical results that the code produces to reference solutions. The classic example of NAV is order-verification, in which one determines whether the numerical solution converges to an exact solution at the same rate as its theoretical order-of-accuracy predicts. If it does not, then incorrect implementation of the relevant numerical algorithm is a strong possibility. Most agree that NAV is an essential step in code verification due to its particularly effective ability to investigate the correctness of algorithm implementations in PDE codes.

Numerical algorithm adequacy verification is concerned with whether or not the algorithm is adequate (in terms of accuracy, robustness, and speed) to solve the mathematical model associated with a particular application such as propulsion modeling within a V&V exercise or within a modeling and simulation project. NAA is important because even if an algorithm has been correctly implemented, it does not necessarily mean that it is adequate for its intended use. Thus it is the application requirements that determine adequacy. It is helpful to keep in mind the distinction between NAV and NAA because it can help sort out the purpose of different tests and determine task priorities within code verification. For a given algorithm, NAV should usually be performed prior to NAA because the adequacy of an algorithm often cannot be determined if it has not been correctly implemented.[2]

[2]If the properties of a numerical algorithm are not mathematically proven, then NAV is precluded for that

It was noted at the beginning of the introduction that the end-product of Verification is an assessment of both the software and the numerical solutions it produces. *At minimum*, the software assessment is made with respect to its use within the V&V exercise itself. This intended use determines which of the capabilities and algorithms of the code will be included within both the NAV and NAA portions of the verification.[3] A narrow, but precise definition of algorithm adequacy derives from the intended use within calculation verification: the collection of algorithms used in one or more of the simulations that are performed for the purpose of the prediction and its solution verification is *adequate* if the algorithms are correctly implemented and, within the memory and efficiency limits of the selected computing platform, the algorithms produce numerical solutions that are both converged and asymptotic. By 'converged' we mean that both the solver and non-linear iteration loops were terminated by satisfying acceptably low relative error tolerances on the residual or on other numerical quantities of interest. By 'asymptotic', we mean that the numerical solution as a function of mesh-discretization size can (hopefully) be shown via mesh refinement studies to be in the asymptotic range of mesh convergence. Under this narrow definition, adequacy is directly determined during the process of creating numerical solutions for the calculation of numerically-precise error estimates and error bars. In practice, 'narrow-NAA' activities are generally considered to be part of the calculation verification process.

More commonly used are broader definitions of NAA [7] that stem from a desire to 'establish confidence that mathematical model and its solution algorithms are working correctly.' Although there are technical SE and CV drivers, another driver of the need for confidence is social and is due to the fact that there is a division of labor in which one group of people function as code developers and another group functions as code users. The first group develops and tests code capabilities while the second group (sometimes called analysts) performs the predictions and their calculation verification. Both groups require evidence that gives them confidence in the code and algorithms. The preferred evidence is frequently based on running the code on *nearby* problems, i.e., ones that are 'close' to the ultimate application. Nearby problems involve a test that resembles the ultimate application in one or more of its essential aspects, but removes non-essential features of the ultimate application so that important behavior can be isolated and the testing is less cumbersome. In this context, passing a series of nearby-problem tests provides evidence that the code is 'working correctly'.[4] Examples of the broader definition of algorithm adequacy include efficiency tests, robustness tests, iterative convergence tests, numerical artifact tests, symmetry tests, benchmark tests [8], and many others. The confidence issue arises in several contexts:

1. In the decision by the users regarding the selection of the code that is to be used in the V&V exercise and beyond,

2. In the hand-off between the code developers and the analysts who will perform calculation verification, and

algorithm. However, the properties of the algorithm can still be explored through computational experiments. Thus, for example, there may be no proof that a given algorithm is second-order accurate, but through computational experiment one can infer the order-of-accuracy by comparing to an exact solution. It may turn out that the algorithm exhibits only first order accuracy. The assessment issue then becomes one of algorithm adequacy or NAA: is a first-order algorithm good enough for the intended application?

[3]This determination can be systematically addressed through code *coverage* tables [5], [6] which explain the mapping between the tests in the verification test suite and the portions of the code that are exercised in a particular prediction simulation.

[4]It should be noted, however, that passing a set of nearby problem tests does not guarantee that the code will then provide an acceptable solution to the simulations to which they are supposedly 'near'.

3. In the subsequent use, improvement, and advertisement of the code after a V&V exercise is completed.

3.0 Code Verification Testing Practices

Testing is an important part of code verification because it is a major activity in the collection of evidence. Various kinds of tests are used in code verification including unit tests, stability tests, robustness tests, tests of numerical options (e.g. patch tests, limiter tests), benchmark tests involving comparison to exact solutions to the PDE, and mesh refinement studies, with or without exact solutions. Some tests can be called NAV-tests since they are geared toward investigation of implementation correctness by showing the results agree with proven mathematical properties of the algorithms. Other tests can be called NAA-tests since they investigate questions related to algorithm adequacy.

One of the best (i.e., most revealing) practices in algorithm verification is to employ tests involving systematic mesh refinement and comparison of the numerical solutions to a predetermined exact solution in order to compute the discretization error when in the asymptotic regime. The acceptance criterion for such a test can be either that (1) the error decreases to zero as the mesh is systematically refined (consistency verification) or (2) the error decreases at the expected rate (order verification, [9]). Assuming the test setup was not flawed, failure to meet the acceptance criteria in tests of this kind generally means that (if the algorithm is well-ordered) the algorithm was not correctly implemented.

A collection of tests forms a test-suite. Tests which are performed for the purpose of code verification are called a verification test-suite (VERTS). Verification test-suites should be well-designed, comprehensive, and maintainable. A *well-designed* verification test-suite is one in which the collective set of tests, when passed, supply compelling evidence to show that the solution algorithms are verified and adequate. A *comprehensive* VERTS is one in which the test-suite properly tests all the codes' features and capabilities with respect to the intended application, and thus ensures there are no coverage gaps.

An important tool in the creation of complete test suites is the Method of Manufactured Solutions (MMS), ([10], [11], [12]). MMS is a form of Order verification in which the exact solution is created in a non-traditional manner. Exact solutions created in the traditional way begin with the statement of the mathematical problem (the PDEs, domain, BC's, coefficients, etc.). Given the problem and associated data, the exact solution is found using classical methods such as separation of variables, series solutions, integral methods, or Green's functions. A major limitation of this approach is that classical solutions can usually be found only by simplification of the mathematical model (e.g., by reducing it to one-dimension, by assuming constant coefficients, by using simple geometric domains, or by linearization). For the purpose of rigorous code verification, this is inadequate because in many of the intended uses of the code, algorithms are used that are more general than what is tested when making these simplifying assumptions. Therefore, if one is restricted to only classical solutions, the test suite will contain significant gaps. The Method of Manufactured Solutions can fill many of these gaps because the method by which the exact solution is created is different. MMS proceeds instead by choosing the exact solutions first and then applying the PDE operators to the chosen solution to find a source term that balances the equations. In this manner, one can create tests to cover the fully-general features and capabilities of the code in terms of showing that the PDEs are solved correctly. MMS is particularly suitable for NAV testing be-

cause realistic exact solutions are not needed for verifying order-of-accuracy or other mathematical properties of the algorithm. MMS can also be valuable as NAA tests, for example, to establish an unknown order of accuracy or to identify algorithmic deficiencies.

MMS is less suited for use in 'nearby problems' due to the difficulty of manufacturing 'nearby' solutions. There are other important open questions in MMS and, more generally, order-verification [13]. For example, there are numerical algorithms used in shock physics, structural dynamics, and fire combustion whose theoretical convergence properties are poorly understood and thus it is unclear in these examples how to interpret a failure to converge to a stationary solution when mesh refinement is employed. There are many other kinds of NAA-tests where MMS is not suitable; an obvious example is the 'testing' that is done in narrow-NAA within SV. Finally, MMS is sometimes criticized due to the occasional necessity of introducing one or more numerical source terms, thus modifying the very code one is to test. This objection is less serious than it might seem because MMS is a self-correcting procedure, i.e., if a coding mistake is introduced into the code due to the addition of a source term, the MMS-test will likely detect it due to MMS tests' extreme sensitivity.

Finally, a *well-maintained* VERTS is a test-suite in which all tests can be run *on-demand*, i.e., the tests will run with the latest release version of the code so that as the code progresses through different versions, one can perform regression-like verification tests to ensure that solution algorithms that were working in earlier release versions also work in the current release version. Further, this activity ensures that the evidence collected to show that the code has been properly verified pertains to the latest code versions and, additionally, that the evidence is available to anyone that needs to see it, whether they be the developers, the code users, or other stake-holders.

A commonly overlooked requirement when developing modeling and simulation software is the need to ensure that the code is verifiable [14]. Five important attributes of verifiable code are: (1) complete documentation of the mathematical model and the solution algorithms, (2) creation of a map linking the mathematical model directly to the software, (3) the ability to input *distributed* source term data for both the interior equations and the boundary conditions to permit MMS tests, (4) the distributed source terms are incorporated into the mathematical model and numerical algorithms, and (5) the ability to automatically compute discretization error in NAV tests by comparing the numerical solution to the exact solution.

4.0 Benefits of Code Verification

Code verification, when done properly, has many benefits. First and foremost, CV plays an important role in V&V because it is an important step in the code-verification, calculation-verification, validation, sensitivity, and uncertainty-quantification sequence. All the activities in this sequence are put at risk if CV is not done or not done well, because incorrect or inadequate algorithms can mean misleading numerical results or even no numerical results at all. Code verification has various other benefits including (i) it assists in the development of improved numerical algorithms, (ii) codes accompanied by evidence of proper verification will be more likely to be used than codes which do not have such evidence, (iii) code verification, when performed as an integral part of code development, ensures that coding mistakes do not mislead algorithm developers as to the adequacy of their algorithms and ensures that their experience over time is based on sound code, (iv) order-verification encourages the use of ordered discretization models and discourages the use of non-ordered approximations, (v) CV increases code/algorithm developer confidence, knowledge, and professionalism, and (vi) it assists in developing the software and analysis products right.

References

[1] *V&V 10 - 2006 Guide for Verification and Validation in Computational Solid Mechanics*, 36p, ASME 2006.

[2] *IEEE Standard for Software Verification and Validation*, IEEE Std. 1012, IEEE Computer Society, 2004.

[3] Bezier, B. *Software Testing Techniques*, International Thompson Press, 1990.

[4] Sommerville, I. *Software Engineering*, Addison-Wesley, 2005.

[5] Dowding, K., Blackwell, B., and Knupp, P. *Demonstrating code verification methods with calore*, SAND2007-5612, Sandia National Laboratories, Albuquerque NM, 2007.

[6] Knupp, P. and Ober, C. *A code-verification evidence-generation process model and checklist*, SAND2008-4832, Sandia National Laboratories, Albuquerque NM, 2008.

[7] Oberkampf, W.L. and Trucano, T.G. *Verification and Validation in Computational Fluid Dynamics*, SAND2002-0529, Sandia National Laboratories, 2002.

[8] Oberkampf, W. and Trucano, T., *Verification and Validation Benchmarks*, SAND2007-0853, Sandia National Laboratories, 2007.

[9] Knupp, P., Ober, C., and Bond, R. *Measuring Progress in Order-Verification within Software Development Projects*, Engineering with Computers, 2007.

[10] Roache, P.J. *Verification of Codes and Calculations*, AIAA Journal, Vol.36, No.5, May 1998

[11] Roache, P.J. *Verification and Validation in Computational Science and Engineering*, Hermosa Publishers, 1998.

[12] Knupp, P. and Salari, K. *Verification of Computer Codes in Computational Science and Engineering*, Chapman & Hall/CRC, Boca Raton FL, 2003.

[13] Oberkampf, W. and Knupp, P. *Open Questions in the Method of Manufactured Solutions*, Internal Memo, March 8, 2004, Sandia National Laboratories.

[14] Roache, P. *Building Codes to be Verifiable and Validable*, p30-38, Computing in Science & Engineering, September/October 2004.